Connect with USBLab

No parallel printer port? Stuart's USBLab plugs into a USB port and provides you with eight bidirectional data bits, three address bits, a read/write direction control line, and a data strobe. It also has a serial input and output. Problem solved.

Back in the good old days of computing, around 1997, you could connect a new widget to a PC's parallel printer port and control it. All you had to do was write to the parallel port's I/O addresses to make things happen. But now most new computers come with either Windows 2000 or XP operating systems, which don't allow user-level programs to access the printer port. You can find a driver that will allow such access, but it will be expensive, complicated to work with, and won't always work if the parallel port is configured in EPP mode. What can you do with a new computer that doesn't have a parallel printer port?

I designed the USBLab to solve these problems. The USBLab plugs into a USB port and provides eight bidirectional data bits, three address bits, a read/write direction control line, and a data strobe. Plus, it has a serial input and output. The former allows you to program the data rate with binary divisor values so you can obtain nonstandard data rates to match your project, which may not use a multiple-of-standard-data-rates crystal. The serial I/O is TTL, not RS-232, so you don't have to add RS-232 conversion to your microcontroller project.

USBLab CIRCUITRY

The USBLab uses an Atmel ATmega8515 microprocessor (U2) connected to a DLP-USB245 (U1) adapter, which is a circuit board mounted to a 24-pin header that will fit in a standard 24-pin DIP socket (see Figure 1). The DLP-USB245, which has a USB connector, connects to a USB port on your PC. The module looks like a standard serial communication device to the PC, but it stores the incoming data and presents it to an external microcontroller (the ATmega8515) via an 8-bit data bus. The DLP-USB245 has read and write input signals so the external microcontroller can send and receive data.

The DLP-USB245 handles the USB protocol, so you don't have to write any of the messy USB protocol software. It also provides low signals on pin 13 (*RXF), when there is incoming USB data to read, and on pin 14 (*TXE), when the transmit FIFO is ready to accept data. The MCU writes to the FIFO just as it would write to a memory device or register by placing data on the eight data lines (pins 17 through 24) and strobing the *WR signal (pin 15) low. Similarly, data is read by strobing *RD (pin 16) low.

The DLP-USB245 uses an FT245BM IC, which is one of a series of the Future Technology Devices International parts that provides modular functional blocks such as serial I/O and parallel FIFO-based interfaces. The DLP-USB245 combines the FT245BM with a crystal, a USB connector, and EEPROM (to store USB enumeration values).

The ATmega8515 microcontroller reads incoming USB data from the DLP-USB245. The interface uses a simple command opcode followed (in some cases) by data. The opcode tells the ATmega8515 whether to write the data to the parallel port, serial port, or data rate registers. In some cases, the opcode isn't followed by data. For example, the command to read the

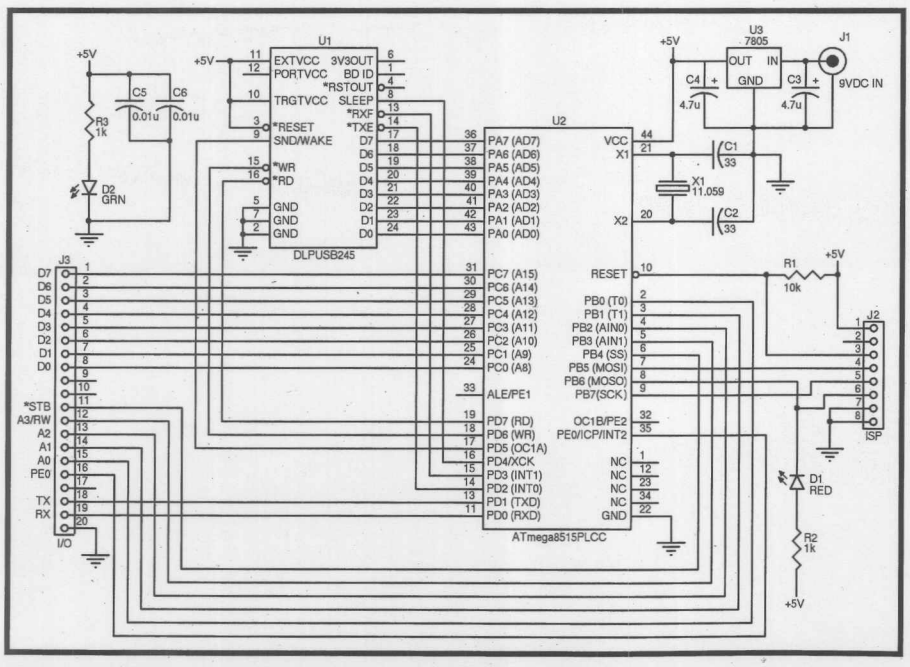

Figure 1—*An ATmega8515 microcontroller controls the USBLab, which requires only three integrated circuits. Connector J2 enables you to in-circuit program the ATmega8515 microcontroller. You may replace J2 with a standard 10-pin connector.*

parallel port doesn't send data; it expects data to be returned.

The ATmega8515 operates from an 11.059-MHz crystal, which provides standard data rates. However, by programming the data rate divisor, you can get nonstandard rates. If you have a specific project for which you want a specific data rate, build the project with a crystal that's a multiple of that rate. The ATmega8515 will operate with a crystal up to 12 MHz.

```
>>>
Board ID: USBLab v1.0
Walking one return code: 4E time: 0.0320000648499
Write addr 0-7 return code: 4E time: 0.0309998989105
Read addr 0-7 return code: 20 23 21 23 22 23 23 23 24 23 25 23 26 23 27 23 4E time:
0.109000205994
PE0 return code: 4E time: 0.140000104904
Serial return data: 30 41 30 42 30 43 30 44 30 45 30 46 30 47 30 48
Set baud Return Data: 4E
Invalid opcode return value: 7F
>>>
```

Figure 2—*The Idle window should show the results of running the TestUSBLab.py program. Timing may vary from these values.*

The circuit is powered by an external supply. A USB device may be powered by the USB bus, but current is limited, and in most cases, the USBLab will be used in conjunction with some kind of self-powered project. To avoid USB power management issues, I made the USBLab self-powered as well.

Power is applied to the USBLab via a coaxial jack (J1) and regulated by a 7805 regulator (U3). A 9-VDC, 500-mA wall wart transformer supplies power. LED D2 indicates when power is applied.

Input/output connector J3 connects to your external project. You can read and write up to eight external registers by decoding the three address lines A0 through A2. Signal PE0 is an extra output signal. Separate commands allow you to set and clear PE0 without affecting the state of D0 through D7 and A0 through A2.

Connector J2 is an eight-pin header that allows the ATmega8515 microcontroller to be programmed in-circuit. This is an in-line header that matches my programmer. You could also use the standard Atmel 10-pin ISP header.

CIRCUIT CONSTRUCTION

I used the PLCC version of the ATmega8515 so the part could be placed in a standard through-hole PLCC socket. The DLP-USB245 was placed in a standard 24-pin, machined-pin IC socket. I don't recommend a leaf-type socket because the DLP-245BM has round pins. You can, of course, wire to the DLP-245BM without using a socket at all.

The expansion connector J3 is a 20-pin (10 × 2) header—the type of header into which you'd plug a ribbon cable. Jack J1 is wired with the center terminal as the positive connection. You can, of course, change that to match a power supply you already have.

C5 is located near the ATmega8515's V_{CC} pin (pin 44). C6 is located near the DLP-USB245 V_{CC} pin (pin 11).

LED D1 should blink when power is applied and the ATmega8515 is programmed. If it doesn't check your connections, make sure that the 5-V supply voltage is correct and that the crystal is connected correctly.

I wrote the TestUSBLab.py checkout program in Python. It uses a communication module called commport.py that I wrote for a previous project. You must have Python and the Python win32 extensions installed on your computer to use TestUSBLab.py.

After installing Python and the Win32 extensions, turn on power to the board and plug it into the USB port. Windows should detect the device. If you're using Windows XP SP1, the DLP communication port driver will probably load automatically. If it doesn't, you can get the driver from the DLP Design's web site. Use the VCP driver.

Connect the expansion connector's pins 18 and 19 (Tx and Rx) with a jumper. Before running TestUSBLab, you must set the COM port to match your system. You can find the COM port by opening the control panel and selecting System, Hardware, and then Device Manager. You should see a COM port in the Ports window that isn't part of your system. If you aren't sure which one it is, check the device manager before and after you plug in the USB cable.

After you identify the COM port number, right click the TestUSBLab.py file, select Edit with Idle, and find the line that opens COM port 7:

```
ch = c.OpenCommPort(7)
```

Change the 7 to whatever COM port the USBLab enumerates as in your system; it will typically be port 3 or 4. It was 7 on my system because I have a com6 device. (Refer to the USB Enumeration sidebar (p. 78) for more information.) Next, save the file and then select the Edit/Run or Run/Run module (depending on your version of Idle) to run the script. You can run the program by double clicking it, but the results would be displayed in a DOS box that disappears after the program is finished. The Idle window should display the results shown in Figure 2.

The data line starting with "Read addr" may vary, but it should be 20 xx 21 xx, 22 xx... The time to complete the tests will vary as well; however, if you get these results back, you'll know the basic hardware is working.

To verify that the expansion port is wired correctly, copy the walking one's code from TestUSBLab.py into a new Python program (keep the imports and port set-up code), and modify it to delay a couple of seconds (with `time.sleep(2)`) after each write. This will allow you to use a logic probe or DVM to verify that the bits are walking through the pins. Or, if you have one, you can use a logic analyzer like I did.

FIRMWARE

The firmware in the ATmega8515 microcontroller loops continuously and monitors the serial port for incoming data and the DLP-USB245 for incoming USB data. The ATmega8515 microcontroller's data bus is configured to access external memory. Because the DLP-USB245 has no address inputs, the ATmega8515 microcontroller's upper eight address outputs are configured to function as general-purpose I/O lines.

If incoming data is found on the serial port, the firmware will write the SendSerial opcode to the DLP-245, then write the received serial byte. If the DLP-USB245 indicates that USB data has been received, the firmware will read the data and process it. The firmware always expects an opcode to be first. If the opcode is SendSerial, WriteParallel, or SetBaudRate, the firmware expects additional data

Opcode value	Meaning
0x00	NOP
0x10	WriteParallel (bits 0–2 are address to write)
0x20	ReadParallel (bits 0–2 are address to read)
0x30	: SendSerial
0x40	SetBaud
0x44	SetPE0
0x48	ClrPE0
0x4C	TransmitNow
0x4D	GetBoardID
0x4E	ReturnAck

Table 1—*The USBLab interprets the opcodes to perform specified actions. Some opcodes are followed by one or more data bytes.*

bytes. One byte is expected for SendSerial and WriteParallel. Two bytes are expected for SetBaudRate.

The firmware checks the opcodes for validity (see Table 1). If an invalid opcode is encountered, a value of 7F is returned to the host. The USBLab won't hang in that case, but if the invalid opcode is caused by a synchronization problem when sending data to the USBLab, you may get some strange results.

The WriteParallel opcode can range from 0x10 to 0x17. ReadParallel opcode can range from 0x20 to 0x27. The lower 3 bits specify which address on the expansion connector are to be written and read. The SetBaud opcode is followed by the high byte of the data rate and then the low byte of the data rate. These values are written directly to the ATmega8515's data rate registers.

The `GetBoardID` command will cause the USBLab to return the text "USBLab" and the board's revision. The string (all ASCII characters) is terminated with a zero byte. Generally, you shouldn't issue the board ID command while the USBLab is receiving serial data because serial data may be inserted ahead of the board ID string. The board ID is intended to identify the USBLab if you have multiple COM devices connected.

Note that the board ID string won't be interrupted with serial or parallel data. Therefore, if your code looks for the text "USBLab" in the returned USB data and then reads until it finds a zero, you can issue and handle the board ID command at any time. The `ReturnAck` command causes the opcode to be returned to the host. The firmware doesn't buffer USB or serial data. If the DLP-USB245 stops accepting data, the firmware will hang and wait for the transmit-ready signal to activate. The firmware object code is contained in the USBLab.hex file. It's an Intel hex format file.

SYSTEM AT WORK

The commport.py code provides simple communication functions that can be used with the USBLab. It's important that the timeout is set like it is in the TestUSBLab.py code because this allows the code to read from the USBLab without having to wait too long.

When processing returned data from the USBLab, you need to process 1 byte at a time because the opcode/data pairs may be broken up into two USB transmissions. In general, you should read an opcode and wait for another byte if no more data is available. You can't guarantee that a single read from the USB COM port will return both bytes of a pair.

I/O timing using the USBLab is a bit different from timing using a parallel port. With the parallel port, your code can manipulate the data bits in line and know when the actual bit changes occur. With the USBLab, the USB interface introduces unknown latency in between writing data and the actual changes on the expansion connector outputs. In addition, the DLP-USB245 has a latency timer for determining how often to transmit stored data to the PC.

You can synchronize the USBLab with your code with the `ReturnAck` command. The USBLab processes commands in the order they're received. By executing a series of parallel write commands followed by a `ReturnAck` command, you can be certain that all the parallel data is written when the `ReturnAck` command is received.

You can do something similar with the send serial command by placing a `ReturnAck` command at the end of the last serial transmit command pair. In this case, the `ReturnAck` command will indicate that the last serial byte was written to the output UART, not necessarily that it was transmitted.

Matching the timing of the serial data with the timing of control inputs is often a problem when you're using a serial port to debug a

USB Enumeration

After a Windows-based PC detects that a USB device has been inserted, it queries the device to see what it is. The query process involves passing several descriptors back and forth, but the result is that Windows obtains a vendor ID (VID) and product ID (PID). These values are used to select the driver that will communicate with the device. Enumeration is the process of obtaining the VID/PID and other information. If Windows can't find a driver, it will ask you for one. The default VID and PID for the DLP-USB245 are 0403 and 6001.

A USB device can use its own driver, or it can enumerate itself as a standard, Windows-recognized device class. These classes include USB mice, keyboards, and mass storage devices. Devices using predefined USB classes are expected to operate according to the protocols for that class, so no driver is needed (or, more precisely, Windows has built-in drivers for them). This is how all the different USB flash memory-based drives can operate without driver software. It's also why those flash memory drives need a driver for Windows 98, which didn't have a predefined mass storage class.

With the right firmware, one device can make use of more than one driver by issuing different VID/PID pairs during enumeration. You could make a device appear as a COM port. Then, by flipping a switch, you could re-enumerate it as a memory-storage device. In fact, so-called "soft" USB devices do exactly that. A soft USB device doesn't contain any permanent storage. The operating code, which is contained in RAM, is lost when you unplug the device or turn off the power. A soft USB device uses a driver that loads the operating code before the device reenumerates as whatever type of device it's supposed to be.

Writing your own USB driver can be a problem because you have to obtain a VID/PID from the USB organization. If you don't, you risk a conflict with a commercial product. USB is intended for products that are produced to the tune of thousands, not one-time experimenter projects. The simplest solution is to use a product such as the DLP-USB245, which comes with drivers, and adapt it to your needs.

project. The USBLab writes received serial data to the DLP-USB245 as it's received, so you can correlate the received serial data with the commands you're sending to the parallel port. To correlate received serial data with the WriteParallel and Set/ClearPE0 opcodes, use the `ReturnAck` command to indicate completion of those commands.

When connecting devices to the expansion connector, make sure you drive data on the data lines only when the USBLab is reading. Use the read/write signal and the data strobe to ensure this. Drive the data bus only when read/write is high and the data strobe is low.

If you fall out of synchronization with the USBLab when you're debugging, your code or the USBLab (or both) will probably confuse the data and opcodes. If that happens, you should reset the USBLab or send a string of NOP or `ReturnAck` commands.

If you don't want to use the USBLab as a COM port, DLP Design has a DLL-based driver that provides essentially the same features. You can call the DLL from any software that's capable of calling a DLL, including Python and, of course, C language. If you're using the DLL driver, you can change the EEPROM values, including the vendor VID and PID, and the latency timer value.

EASY USB

The USBLab provides an easy way to access functionality similar to the old parallel port, but on newer computers and operating systems. In addition, the inclusion of the serial port provides new debug potential for your computer-connected designs. Finally, you can apply the principles described in this article to adapt the DLP-USB245 to your own application. Now you can add an easy USB connection to your next design.

Stuart Ball is an engineer at Seagate Technologies with more than 20 years of experience in embedded systems. He has a B.S.E.E. from the University of Missouri-Columbia and an M.B.A. from Regis University in Denver. Stuart has authored three books about embedded systems. You may contact him at stuart@stuartball.com.

PROJECT FILES

To download the code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2005/178.

SOURCES

ATmega8515 Microcontroller
Atmel Corp.
www.atmel.com

DLP-USB245 USB Adapter
DLP Design
www.dlpdesign.com

FT245BM USB FIFO IC
Future Technology Devices International
www.ftdichip.com

Win32 Extensions
Mark Hammond
http://starship.python.net/crew/mhammond/win32/Downloads.html

Python scripting language
Python Software Foundation
www.python.org